Module 05: Software Security

Week 10& Week 11: Memory Corruption & Runtime Attacks

Daniele Lain, Claudio Anliker, Giovanni Camurati, and Srdjan Čapkun

daniele.lain@inf.ethz.ch, claudio.anliker@inf.ethz.ch, giovanni.camurati@inf.ethz.ch, srdjan.capkun@inf.ethz.ch

November 2021

Welcome to Module 05 of the Information Security Lab. This module is about software vulnerabilities, in particular runtime attacks. We begin with a few pointers and instructions. The exercise and lab sessions will both be held in person. The exercise sessions can be joined remotely via Zoom too. Please join the Zoom session via the links provided in Moodle.

Exercise Sessions

The first exercise session will consist of a general presentation of the objectives and an introduction to the graded assignment and its evaluation criteria. We will also give an introduction to some of the tools that are helpful for this lab.

There will be no exercise session in the second week of the module (Week 11).

Lab Sessions

The lab sessions are intended to discuss specific problems that you might be having with the lab. You are also welcome to post your questions and to join ongoing discussions on the Moodle forum for the module. Please make sure that you do not share solutions when asking or answering questions.

Overview

This lab is focused on understanding *memory corruption* vulnerabilities. These vulnerabilities allow specially crafted attacker inputs to tamper with a vulnerable software by changing and overwriting the software's memory in its stack or heap. For example, we can modify program data (e.g., change variables), or change control data (e.g., return addresses) to hijack the software's control flow and make the software execute injected code.

Please note that we expect you to use `Python 3` for this lab. We will not accept submissions/solutions written in any other language, including older versions of Python. *So please make sure that your submissions work with Python 3.*

Getting Started

We provide a Virtualbox virtual machine in which all exploits need to work. User and password are syssec and syssec.

You can download the VM from:
`https://polybox.ethz.ch/index.php/s/sN9cYH4QiMooFr6/download`.

You can download the files for this exercise from this link:
`https://polybox.ethz.ch/index.php/s/wbEO8wBSkaAVpXD`

In the folder pointed by the second link, you will find an archive labeled with your student ID. The archive contains executables with different mitigations and vulnerabilities that loosely follow the history of memory corruption techniques. *The provided executables are personalized, so make sure to use the executables from the archive labeled with your student ID. Otherwise, your solutions may not be accepted.*

You will need to write exploits for each of the executable that achieve different goals, usually hijacking its control flow to retrieve the content of a secret file, called *flag*.

The folder also contains a `check_environment.py` script that you can use to double-check if your VM environment is well configured.

Exploits. The goal of your exploit is written in the description of every challenge. All different goals involve having the executable print out the content of the *flag* file, that is alongside the executable. Your exploit should print out this content. Your exploits do not need to make the executable exit cleanly: for example, it is fine if after the goal of the exploit was achieved, the program exits with a segmentation fault. Further, your exploit are also allowed to print more information to screen, as long as the content of the *flag* file is also printed at any point in the output. For each exercise, you are expected to create a Python script that exploits the binary to achieve the listed goal.

Virtual Machine We will distribute a virtual machine based on Ubuntu 20.04 with some of the tools mentioned here already installed and configured. The virtual machine is your reference environment: your exploits need to work on the VM, with the same `libc` that is in the VM (be careful not to update it).

`pwntools`. As mentioned above, you need to write your exploit in Python 3. To help you, you can use `pwntools`, a Python library that simplifies interactions with binary files. `pwntools` allows you to execute a binary, send and receive data to it as if you were writing on a shell and reading from it, and it simplifies calculation and conversion of addresses and offsets. You can find its documentation here [1]. Thus, since your script has to print out the content of the *flag* file when it reads it, your workflow will typically involve sending data to the program (e.g., with `sendline` or `send`), receiving its output (i.e., what it prints out on `STDOUT`), and when this output finally contains the flag, printing it to screen.

ASLR. Some of the exercises need to be run with ASLR off. To do so, if you use `pwntools` and `pwntemplate` (like the provided skeletons) you can pass the `NOASLR` argument to the exploit script, e.g., `python3 exploit1b.py NOASLR`.

Reverse-engineering and studying the executables. To ease understanding what the executables do, you might need to (i) disassemble and/or decompile the executable; and (ii) run it with a debugging tool that helps you see the memory layout and execute it step-by-step.

To (i) disassemble and decompile the executables, you can use a interactive disassembler software such as Ghidra. While disassembling, i.e., reconstructing the assembly code of the binary, we recommend using Ghidra, because it also features an experimental *decompiler*. A decompiler tries to reconstruct a reasonable interpretation of the original source code, either in code like C, or in pseudo-code. Decompiling the executables will greatly help you reverse-engineer them.

To (ii) run the executable to see the layout of the memory (e.g., of the stack), and the value of local and global variables, you may use a debugger tool such as GDB. A debugger tool allows to set *breakpoints* on instructions. A breakpoint stops the executable when it reaches the target instruction, and allows you to examine the memory at your desired point of execution. We recommend using GDB with the GEF plugin, both pre-installed in the provided VM.

If you pass the `GDB` argument to a `pwntools` exploit (like the skeletons we provided), `pwntools` will take care of running the executable under GDB. Furthermore, the exploit contains the `gdbscript` variable, that you can populate with GDB commands you would like to execute. For example, to set two breakpoints, one at the start of `main` and one at the start of `check_authorization`, you would add two lines in `gdbscript`: `b *main` and `b *check_authorization`. Then, you can launch your exploit with the GDB argument, e.g., `python3 exploit.py GDB`, and this will run the executable under GDB, run your exploit in it, and break at the start of `main` and `check_authorization`.

Important: launching the executables from GDB *might* change the addresses of the stack, local functions, and library functions, *even on non-PIE, not ASLR-protected executables*. It's fine to use these addresses while developing your exploits and using GDB to do so – however your final exploit should run *outside* of GDB. To get the addresses of, e.g., buffers and functions outside GDB, you can start the executable and then `attach` GDB to its process using `sudo gdb --pid=XYZ` – this will allow you to see the memory layout without the offsets introduced by GDB.

Part 1 - Simple Overflows

The first two executable files you have to work with are meant to show you how to leverage an overflow to write controlled data on the stack.

Exercise 1a - Changing a variable [Ungraded Example]

This first simple exercise shows you how to change values of other variables on the stack thanks to a buffer overflow. This exercise is not graded and will be used as an introductory example in the exercise session.

Goal. Your exploit needs to trick the executable into calling the `uncallable` function by taking the correct `if` path in the `main` function. Do this by changing the value of the variable that `check_authorization` returns. Observe: can you overflow something in the `check_authorization` function? Are the buffer you can overflow and the variable that is returned by `check_authorization` neighboring in the stack's memory?

Info: The 64 bit executable is not PIE, has no canaries. Stack is `W^X`. Should be run with ASLR off.

Exercise 1b - Changing return address

This second simple exercise has a slight change compared to 1a: now the `uncallable` function is *never* called, so changing the return value would not help.

Goal. Your exploit needs to make the executable call the `uncallable` function. Do so by changing the return address of the `check_authorization` function so that it returns to `uncallable` instead of the previous location in the `main`.

Info: The 64 bit executable is not PIE, has no canaries. Stack is `W^X`. Should be run with ASLR off.

Part 2 - Shellcoding

As you saw in the lecture, until 2003 memory pages on the stack used to be executable. The immediate danger posed by this design is that attackers with an overflow on a vulnerable buffer could write their own code in the stack, and change the return address to make the program return to their own code and execute it. A piece of code that the attacker injects and spawns a shell is called a *shellcode*.

Exercise 2a - Simple Shellcode

This first exercise offers you a big buffer where there's plenty of space for your shellcode.

Goal. Your exploit needs to inject and jump (i.e., return to) some code that ultimately allows you to read the `flag` file. You can inject a complete shellcode and use the spawned shell to read the flag file (e.g., with `cat flag`), or a piece of code that only reads the flag file. To generate some shellcode, you can find plenty of sources and tools [2], or you can refer to the dedicated `pwntools` module [3], that can help you easily generate one.

Info: The 64 bit executable is not PIE, has no canaries. Stack is executable. Should be run with ASLR off.

Exercise 2b - Small Buffer Shellcode

This exercise is similar to the previous one, but now the buffer, and thus the space from the start of the buffer to the return address of the `check_authorization` function is very small.

Goal. Your exploit needs again to inject and return to your (shell)code, and print out the content of the `flag` file. You need to find a solution to work against the small buffer space.

Info: The 64 bit executable is not PIE, has no canaries. Stack is executable. Should be run with ASLR off.

Part 3 - Leaks

We continue our exploration of memory corruption vulnerabilities and their mitigations: now `W^X` is enabled, and we cannot inject and execute our own code anymore. We will need to *reuse* code that is already in the binary and its linked libraries; however, first we will understand how to leverage *leaks* of memory from the stack, and use these leaks to prepare our exploits and defeat common mitigations.

Exercise 3a - Stack Canaries

We saw in the lecture that one of the first mitigations against buffer overflows is the use of *stack canaries* – random values that protect the return address of functions from being overwritten by an overflow. Unfortunately, a canary can be defeated by vulnerabilities that leak the portion of the stack containing its value.

Goal. Your exploit, similarly to Exercise 1b, needs to call the `uncallable` function. To do so, you need to find a way to leak the canary that protects the return address of `check_authorization` and rewrite it when you overwrite the return address. Observe carefully the overflow in this exercise: it is different from the overflow in Part 1 and Part 2. This new overflow allows you to overwrite arbitrary memory and does not automatically terminate the strings – recall that functions that print strings, such as `printf` assume that a string is terminated by a `NULL` byte, i.e., `\x00`.

Info: The 64 bit executable is not PIE. Stack is `W^X`. Should be run with ASLR off.

Exercise 3b - ASLR of local functions

So far, all exercises had fixed memory locations and addresses both for libraries and the program functions and stack itself. This exercise enables Address Space Layout Randomization (ASLR), meaning that the base address of libraries (such as libc), program functions (such as `uncallable`) and the stack itself change at every execution. We can easily see how this mitigation would break our previous exploits for, e.g., Exercise 3a and Exercise 3b: we cannot know statically the correct address to return to. ASLR can be partially defeated if there's a leak of at least one memory address of, e.g., a function: since ASLR displaces by a *single* random offset, knowing the runtime address of one function of the program and the static addresses of the other functions of the program is enough to break randomization.

Goal. Your exploit needs to call the `uncallable` function. To do so, you need to compute the *runtime* address of `uncallable`: if you can leak the *runtime* address of any function of the program, you can compute the runtime address of `uncallable` by computing the offset between the two functions, and knowing that this offset dos not change when ASLR is in place.

Info: The 64 bit executable is PIE. Stack is `W^X`. Should be run with ASLR on.

Part 4 - Code Reuse

As hinted, `W^X` defeats code injection - but does not prevent *code reuse*. If we can find interesting functions that are loaded by the program and we can redirect control flow by, e.g., overwriting a function pointer or return address, we can reuse (parts of) such functions for our nefarious purposes.

We will now see two simple forms of code reuse: *return-to-system* on 32-bit and 64-bit systems, where we redirect control to the `system` function in `libc` - we just need to make sure to pass parameters to it in the right way. For this, you will need to understand the *calling conventions* of `x86` and `x86-64`, for which you can find plenty of explanations [4] online.

Exercise 4a - Return-to-libc on 32-bit

The calling convention of x86 requires parameters to be passed on the stack.

Goal. Your goal is to execute `system(“cat flag”)` by changing the return address of `check_authorization`. You need to prepare the stack correctly for the call to `system`, in particular passing the correct address of the argument (i.e., the address of the string `“cat flag”`, that you have to write somewhere in the stack) in the expected stack position.

Info: The 32-bit executable is not PIE. Stack is `W^X`. Should be run with ASLR off.

Exercise 4b - Return-to-libc on 64-bit

The calling convention of x86-64 has a twist: the first few function parameters are now passed via *registers* - so you need to be able to load the content you would like as parameters on registers, by using existing code in the binary.

Goal. Your goal is to execute `system(“cat flag”)` by changing the return address of `check_authorization`. You need to prepare the environment correctly for the call to `system`, in particular loading the address of the argument (i.e., the address of the string `“cat flag”`, that you need to write somewhere in the stack) in the correct register used in x86-64 Linux to pass the first parameter to function calls.

Info: The 64-bit executable is not PIE. Stack is `W^X`. Should be run with ASLR off.

Exercise 4c - Custom ROP-Chain

In this exercise, your binary does not contain the string `cat flag`, meaning that you cannot utilize its address to prepare the stack for a call to `system`. This reflects a much more realistic scenario, where you only have a buffer overflow vulnerability to start with and have to prepare the parameters for your call to `system` yourself. This can be done with return-oriented programming (ROP), which is a more generic approach of what you have seen in the last two exercises. Instead of a single call to `system`, you might need to prepare the stack for several calls to different *gadgets*. A gadget is a set of one of several useful instructions in executable memory that are followed by a `ret` instruction. Providing addresses of such gadgets in the stack can enable you to return from one gadget directly to the next, and thus put them together to produce the code you want to execute. Although this concept might seem a bit difficult on first glance, the underlying concept is fairly simple. You will find plenty of information on ROP online.

Goal. Your goal is again to execute `system(“cat flag”)`, this time by changing the return address of `get_message()`. Coming up with a ROP chain by yourself can be tedious, we recommend that you look into ways to find a chain automatically, e.g. using `ropper`.

Info: The 64-bit executable is not PIE. Stack is `W^X`. Should be run with ASLR off.

Part 5 - Format Strings

As you saw in the lectures, format string vulnerabilities allow the attacker to read or write from memory. A detailed explanation of format strings can be found in [7] and [8].

Warning: Our exercises are on 64 bits and are not a mere application of the classic examples for reading/writing memory at an arbitrary address provided by the attacker. However, if you understand the principle of how format strings work, writing the exploits becomes fairly simple.

`Hint:` do not try to provide the 64-bit address you want to read/write.

Exercise 5a - Format String (Read)

In the previous exercises, you saw how to defeat a stack canary protection using a leak. In this exercise you will do the same by leveraging a format string vulnerability.

Goal. As usual, your goal is to execute the `uncallable` function that prints the flag. After having identified a format string on which you have control, try to use it to leak the value of the canary. Where is the value of the canary located? Is it passed as paramter to a function? Is it close to another known value? Once you have found the canary, you will be able to continue with a buffer overflow exploit. Since the goal is to focus on the format string vulnerability, the exploit is easier than previous exercises and it does not require to modify the return address.

Info: The 64-bit executable is not PIE. Should be run with ASLR off.

Exercise 5b - Format String (Write)

In this exercise we will see that format strings can also be exploited to write memory. In Exericise 4a you have used a format string to leak canary, in this exercise you will use a format sting to modify a variable.

Goal. As usual, your goal is to execute the `uncallable` function that prints the flag. Look at the binary, is there a variable that you can modify to call the `uncallable` function? What is the address of this variable? Where is it stored in memory? Can you write at that address and modify it?

Info: The 64-bit executable is not PIE. Should be run with ASLR off.

Part 6 - Advanced Challenges

Exercise 6a - Snake

Each of the previous exercises focused on a specific vulnerability inserted in a toy program, and you knew in advance which exploit technique you had to use. However, in the real world vulnerabilities are hidden in large code bases and you do not know in advance what they are. In this part we provide you with a more realistic example. Despite being still fairly simple, it is a complete game written in around 250 lines of C code. It contains a vulnerability that you have to exploit. Do not be afraid, the exploit per-se is simple and you have already solved more sophisticated challenges.

The snake game is a very funny game that consist in controlling a snake in a box, with the goal of eating as many fruits as possible. Each time the snake eats a fruit it becomes longer, making the game more difficult. You win when the snake reaches a certain size. Have fun!

Goal. As usual, you need to print the flag by running the `uncallable` function. This time, it is up to you to find how to do that. Try to understand how the program works, what vulnerability it contains, how to exploit it to reach your goal. Hint: this program uses a lot of function pointers. Even a small change to a function pointer could be dangerous. Hint: do not waste time writing a program that finds the right move to go to the next fruit, you can just use the command 'A'.

Info: The 64-bit executable is not PIE.

Exercise 6b - Shellcode with Limited Character Set

In this part, we are building on what you have learned about shellcodes in Part 2. The following exercise contains a small program to manage notes. It is very simplistic, only allowing you to take new notes or display all notes that have been taken so far. The notes themselves are stored in `notes.txt` in a proprietary file format that starts with a predefined file header, i.e. a 32-byte sequence. Consequently, your input must not contain these byte patterns. If it does, it is rejected by the program. This limitation does not harm the program's intended functionality of managing notes, since most printable ASCII characters are allowed. However, it poses additional challenges when writing shellcode. Your job is to come up with a shellcode that works under these additional constraints.

To solve this exercise, we recommend that you do follow these steps:

- Inspect the buffer overflow vulnerability and find the offset to overwrite the return address.
- Find a way to redirect your execution into the buffer.

- Analyze the binary to identify the forbidden characters
- Build an exploit that does not rely on the forbidden characters. We recommend that you use the shellcode from Exercise 2 as a starting point. *Hint:* Think of how you could utilize the executable stack to *build* your shellcode or, more specifically, generate the forbidden characters that you cannot enter directly. You might want to look at the `add`, `sub` instructions as a starting point.

Info: The 64-bit executable is not PIE, it has no canaries. Stack is executable. It should be run with ASLR off.

Goal. You need to exploit the buffer overflow vulnerability by injecting a shellcode that reads the content of `flag`.

Evaluation

We will evaluate your Python scripts against the same personalized executables that we distributed to you, with a **secret flag** owned by root.

Grading: All exercises from Part 1 to Part 5 are worth 1 point. Exercises in Part 6 are more challenging and they are worth 2 points. The total number of points that can be obtained is 14: a passing grade is awarded with 8 points, while the maximum grade is awarded with 12 points. This means you can get a grade 6 on this module even if you skip one of the exercises in Part 6 or two of the exercises in Parts 1 to 5.

- Exercises 1b to 5b: 1 point each.
- Exercise 6a and 6b: 2 points each.
- Points necessary to pass: 8.
- Points necessary for maximum grade: 12.

For each executable, if your script successfully **exploits the binary** to print out the content of the flag file you will get a full point. Your script is not allowed to directly access the flag file and on our evaluation system we will ensure that the script can only access the flag file through a successful exploit. If your exploit fails to print the content of the flag file, takes too long (more than 30 seconds) to finish, or has errors in its execution, we will run it again for a total of 3 times. If in these 3 attempts it does not print the flag, you will not get the point.

Thus, it is important for your exploits to be reliable: remember the exploit should run outside GDB, thus they need to have the correct offsets (not changed by GDB – remember to `attach` to get these addresses).

The exploits will run on the same Virtual Machine that we distributed you, with `libc` version `Ubuntu GLIBC 2.31`. There will be Python 3 with `pwntools`, but other

Python packages are not guaranteed to be installed – only the ones in your VM as it was distributed. The VM will not have any network connectivity.

The exact commands that we will run are:

- Exercise 1b: `python3 exploit1b.py NOASLR`
- Exercise 2a: `python3 exploit2a.py NOASLR`
- Exercise 2b: `python3 exploit2b.py NOASLR`
- Exercise 3a: `python3 exploit3a.py NOASLR`
- Exercise 3b: `python3 exploit3b.py`
- Exercise 4a: `python3 exploit4a.py NOASLR`
- Exercise 4b: `python3 exploit4b.py NOASLR`
- Exercise 4c: `python3 exploit4c.py NOASLR`
- Exercise 5a: `python3 exploit5a.py NOASLR`
- Exercise 5b: `python3 exploit5b.py NOASLR`
- Exercise 6a: `python3 exploit6a.py NOASLR`
- Exercise 6b: `python3 exploit6b.py`

Submission Format

Submit a zip file containing all your exploit files in a **flat structure**: your zip archive should only contain the files `exploit1b.py`, `exploit2a.py`, ..., `exploit6b.py`.

In conclusion, *happy hacking*!

References

1. `http://docs.pwntools.com/en/stable/`
2. `http://shell-storm.org/shellcode/`
3. `http://docs.pwntools.com/en/stable/shellcraft/amd64.html`
4. `https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl`
5. `https://github.com/david942j/one_gadget]`
6. `https://refspecs.linuxbase.org/LSB_3.1.0/LSB-generic/LSB-generic/baselib---libc-start-main-.html`
7. `https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf`
8. `http://phrack.org/issues/59/7.html`